Second concours de l'ÉNS de Lyon
Épreuve d'informatique

ÉNS de Lyon

45 minutes – 11 juillet 2007

Avertissement et conseils

- *Les questions sont données en italique.*
- *Les algorithmes seront décrits dans un langage au choix du candidat.*
- *L'énoncé proposé n'est pas nécessairement conçu pour être totalement résolu dans le temps imparti. Le candidat cherchera en priorité à élaborer une réponse de qualité à une ou plusieurs des questions des parties 1 à 3. Il sera tenu compte de la rigueur des réponses dans l'évaluation de l'épreuve.*
- *L'exercice pourra déboucher sur des questions supplémentaires posées au candidat après la fin de la présentation des réponses préparées.*

Satisfaire des formules booléennes

On s'intéresse dans cet énoncé à la recherche de solutions à des formules booléennes.

1 Formules booléennes

1.1 Introduction à l'algèbre de Boole

On s'intéresse à l'algèbre de Boole suivante : les éléments appartiennent à $\{\text{True}, \text{False}\}$ et les opérations sont ET, noté $\wedge$, OU noté $\vee$ et NON noté $\neg$, dont les tables de vérité sont données ci-dessous :

$\neg$: $\neg\text{False} = \text{True}$ et $\neg\text{True} = \text{False}$

$\vee$:

	False	True
False	False	True
True	True	True

et enfin $\wedge$:

	False	True
False	False	False
True	False	True

Les opérations $\wedge$ et $\vee$ sont associatives et commutatives. Elles vérifient $a \wedge \neg a = \text{False}$, $a \vee \neg a = \text{True}$. Elles sont distributives l'une par rapport à l'autre : $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ et aussi $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$. Enfin on a des règles d'absorption : $a \vee (a \wedge b) = a$ et $a \wedge (a \vee b) = a$, mentionnées dans un souci d'être complet mais qui ne devraient pas nous servir.

1.2 Formule booléenne

Définition. Une formule booléenne f à n variables $x_1, \dots x_n$ est une expression formée à l'aide des opérations $\wedge$, $\vee$ et $\neg$ et faisant intervenir les n variables x_i pour i compris entre 1 et n.
Par exemple $f(x_1, x_2, x_3) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (\neg x_3)$ est une formule booléenne à 3 variables. On évalue f sur le triplet $(\text{True}, \text{False}, \text{True})$ à True, en effet $f(\text{True}, \text{False}, \text{True}) = (\text{True} \wedge \text{True}) \vee (\text{False} \wedge \text{False}) \vee \text{False} = \text{True} \vee \text{False} \vee \text{False} = \text{True}$.

Pour simplifier les exercices, on suppose désormais que toute formule booléenne a n variables, avec n une constante de nos programmes.

Question 1. *Soit f une formule booléenne à n variables, autrement dit une fonction de n variables booléennes dans $\{\text{False}, \text{True}\}$, quel est le cardinal de l'ensemble de définition de f ?*
On considère à nouveau l'exemple $f(x_1, x_2, x_3) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (\neg x_3)$. Représentez sur un dessin à 3 dimensions l'ensemble des valeurs possibles pour x_1, x_2, x_3. Marquez les valeurs pour lesquelles $f(x_1, x_2, x_3) = \text{True}$.

On dit que ces valeurs rendent la formule f vraie, ou encore qu'elles satisfont f. On va chercher par la suite à déterminer si une formule est satisfaisable (*satisfiable* en anglais) en exhibant une solution qui la satisfait, ou si elle ne l'est pas.

2 Formule booléenne en forme normale CNF

Par souci de simplification, on écrit toutes les formules booléennes de façon normalisée, la CNF : *Conjunctive Normal Form* ou forme normale conjonctive en français.

Définition. Une formule booléenne f est en CNF si f s'écrit

$$f(x_1, \dots x_n) = t_1(x_1, \dots x_n) \wedge t_2(x_1, \dots x_n) \wedge \cdots \wedge t_m(x_1, \dots x_n)$$

où chaque terme t_i, appelé *clause*, est une disjonction de certaines variables x_j pour $j \in J_i$ ou de leur négation $\neg x_j$ pour $j \in J_i'$:

$$t_i(x_1, \dots x_n) = \bigvee_{j \in J_i} x_j \vee \bigvee_{j \in J_i'} \neg x_j.$$

Question 2. *Mettez sous forme CNF la formule* $f(x_1, x_2, x_3) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (\neg x_3)$.
Montrez que toute formule booléenne peut s'écrire en CNF.

On va maintenant définir un type de données pour représenter les formules booléennes en forme CNF. Une formule booléenne f est une conjonction de clauses, chaque clause étant une disjonction. Si

$$f(x_1, \dots x_n) = t_1(x_1, \dots x_n) \wedge t_2(x_1, \dots x_n) \wedge \dots \wedge t_m(x_1, \dots x_n) = \bigwedge_{i=1}^{m} t_i(x_1, \dots c_n),$$

on propose de représenter chaque clause t par un tableau T à n entrées de la façon suivante :
- si $t(x_1, \dots x_n)$ contient x_j (c'est-à-dire si $j \in J_i$) alors $T(j) = 1$,
- si $t(x_1, \dots x_n)$ contient $\neg x_j$ (c'est-à-dire si $j \in J'_i$) alors $T(j) = 0$,
- si $t(x_1, \dots x_n)$ ne contient pas du tout la variable x_j (c'est-à-dire si $j \notin J_i$ et $j \notin J'_i$) alors $T(j) =$absent.

Question 3. *Définissez le type des éléments d'un tel tableau, que vous appellerez* `elt_type`.
Définissez le type d'un tel tableau T, que vous appellerez `clause_type`.
On représentera la formule booléenne f comme une liste de clauses, proposez un type pour f, que vous appellerez `formula_type`.

On suppose que l'on dispose d'une fonction qui prend en entrée une formule f, un tableau de n valeurs booléennes `var[1..n] of boolean` qui contient des valeurs pour les variables $x_1, \dots x_n$, qui évalue f en ce n-uplet de valeurs booléennes et qui retourne True ou False selon que f(`var`) s'évalue en True ou False : l'en-tête d'une telle fonction pourrait être
`function eval (formula_type : f, var[1..n] of boolean : val) return boolean`
On ne demande pas (pour le moment) d'écrire une telle fonction.

3 Satisfaire des formules booléennes en CNF

On cherche à déterminer si la formule f est satisfaisable. Pour cela on cherche à déterminer un n-uplet de valeurs booléennes qui satisfait f, s'il en existe, ou à démontrer qu'il n'en existe pas.

On suppose que l'on dispose également d'une fonction qui prend en entrée un n-uplet de variables booléennes et qui retourne "le suivant", de façon à énumérer tous les n-uplets, et qui cycle quand tous ont été épuisés :
`function suivant (var[1..n] of boolean : courant) return var[1..n] of boolean`

Question 4. *Écrivez un algorithme qui prend en entrée une formule f à n variables et qui retourne une valeur booléenne indiquant si f est satisfaisable.*

Quelle est la complexité de cet algorithme ? On comptera pour une opération élémentaire l'appel à la fonction `eval`.
Quelle est la complexité de la résolution du problème de satisfaisabilité basée sur cet algorithme ?

On sait qu'il n'est pas possible d'obtenir un algorithme théoriquement plus rapide. On cherche cependant des techniques pour réduire cette énumération exhaustive : même si dans le pire des cas on devra procéder à une énumération complète, on espère pouvoir accélérer, dans les cas pratiques, la détermination de solutions si la formule est satisfaisable, ou la certitude qu'elle n'est pas satisfaisable le cas échéant.

On va parcourir la formule pour déterminer si une valeur d'une des variables booléennes, x_j, est fixée par l'une des clauses : une variable x_j est fixée à True si l'une des clauses t_i s'écrit x_j, à False si $t_i = \neg x_j$ et elle reste indéterminée sinon. Sur notre exemple $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (\neg x_3)$, x_3 est fixée à False par la dernière clause. Si une variable est fixée à True par une clause et par False par une autre, cela implique que la formule n'est pas satisfaisable.

Après une passe sur la formule qui a indiqué que la variable x_j était fixée par la clause t_i, on met à jour les autres clauses où x_j apparaît :
- si t_k ne contient aucune occurrence de x_j alors la clause t_k n'est pas modifiée,
- si après avoir remplacé x_j par sa valeur dans t_k, la clause t_k devient True $\vee \ldots$ alors t_k disparaît de la formule,
- si après avoir remplacé x_j par sa valeur dans t_k, la clause t_k devient False $\vee \ldots$ alors x_j disparaît de t_k,
- si après avoir remplacé x_j par sa valeur dans t_k, la clause t_k devient False alors la formule n'est pas satisfaisable.

Par exemple, si

$$\begin{aligned} f(x_1,x_2,x_3) &= t_1(x_1,x_2,x_3) \wedge t_2(x_1,x_2,x_3) \wedge t_3(x_1,x_2,x_3) \wedge t_4(x_1,x_2,x_3) \wedge t_5(x_1,x_2,x_3) \\ \text{avec} & \\ t_1(x_1,x_2,x_3) &= x_1 \vee \neg x_2 \vee x_3 \\ t_2(x_1,x_2,x_3) &= \neg x_3 \\ t_3(x_1,x_2,x_3) &= \neg x_1 \vee x_2 \\ t_4(x_1,x_2,x_3) &= x_2 \vee \neg x_3 \\ t_5(x_1,x_2,x_3) &= x_1 \vee x_3 \end{aligned}$$

La clause t_2 fixe la variable x_3 à False, t_1 devient $t_1 = x_1 \vee \neg x_2 \vee \text{False} = x_1 \vee \neg x_2$, t_2 devient True, la clause t_3 n'est pas modifiée, t_4 devient $t_4 = x_2 \vee \text{True} = \text{True}$ et enfin t_5 devient $x_1 \vee \text{False} = x_1$. La formule f devient donc

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge x_1.$$

On peut alors recommencer : la variable x_1 est fixée par la dernière clause.

Question 5. *On suppose que la valeur de la variable x_j a été fixée. Écrivez la mise à jour d'une clause t_i. Écrivez ensuite la mise à jour de la formule f.*

On voit sur l'exemple qu'on peut réitérer cette procédure et qu'une passe peut conduire à fixer d'autres variables.

Question 6. *Écrivez l'algorithme qui répète cette recherche de variables dont la valeur est fixée autant de fois que possible, et qui s'arrête soit si la formule n'est pas satisfaisable, soit si elle est satisfaisable, soit si elle est simplifiée autant que possible.*

Question 7. *Réécrivez l'algorithme d'énumération exhaustive qui commence par appliquer cet algorithme de simplification, puis qui tient compte des simplifications apportées pour déterminee soit que la formule n'est pas satisfaisable, soit les solutions qui satisfont cette formule.*

4 Pour aller plus loin...

Question 8. *Écrivez la procédure* `eval` *mentionnée plus haut.*

Question 9. *Que fait l'algorithme* `mystere` *donné ci-dessous ?*
Quelle est sa complexité ?
Quelle est la probabilité qu'il réponde `echec` *?*

```
function mystere (f: formula_type; m,n:integer; epsilon: real) return reponse
  //m: nombre de clauses, n: nombre de variables
  {
  i,nb_loop : integer;

  nb_loop := ceil( log(epsilon) / log(1 - (1-2^(-n))^m) );
  for i = 1 to nb_loop loop
    generer sol, un n-uplet de valeurs booleennes pour x(1) ... x(n);
    if ( eval(f, sol) == True) then
      return sol;
    end if;
  end for;
  return inconnu;
  }
```

5 Pour en savoir plus...

Le problème de satisfaisabilité est connu en informatique théorique sous le nom de SAT. Il s'agit d'un problème NP-complet.

En théorie de la complexité, la classe P (P pour Polynomial) consiste en tous les problèmes qui peuvent être résolus par une machine déterministe (par opposition à probabiliste) en temps polynomial en la taille des entrées (pour nous, le nombre n de variables et le nombre m de clauses). La classe NP (NP pour Non-déterministe Polynomial) est la classe des problèmes dont on peut vérifier une solution en temps polynomial sur une machine séquentielle déterministe, ou de façon équivalente dont on peut trouver une solution en temps polynomial sur une machine non déterministe. Par exemple, factoriser un entier en produit de facteurs premiers est peut-être difficile (on ne le sait pas encore), en revanche vérifier une factorisation est facile (il suffit d'effectuer une multiplication et de me croire quand j'affirme que tester la primalité est dans P). L'une des plus grandes questions ouvertes en informatique théorique est la suivante :

$$P = NP?$$

On ne connaît pas encore la réponse à cette question (et ce problème ne l'a pas résolu). Le Clay Mathematics Institute a offert un prix de 1 million de dollars pour la première

réponse (avec preuve !) correcte.

Pour aborder cette question $P = NP$?, le concept de NP-complétude est très utilisé. Les problèmes NP-complets sont les problèmes les plus difficiles de NP, ils sont donc les problèmes candidats à ne pas appartenir à P si jamais $P \neq NP$. On définit les problèmes NP-durs comme étant les problèmes auxquels tout problème de NP peut se ramener par un algorithme s'exécutant en temps polynomial. Les problèmes NP-complets sont les problèmes NP-durs qui appartiennent à NP. En particulier, notre problème SAT est un problème NP-complet. Cela signifie que si l'on démontrait que SAT appartient aussi à P, alors on aurait démontré que $P = NP$. Actuellement, on n'a pas réussi à trouver un algorithme rapide pour SAT...

SAT a été le premier problème naturel pour lequel on a montré la NP-complétude. Ce résultat a été établi par Stephen Cook en 1971 et est désormais connu sous le nom de théorème de Cook. La preuve était assez ardue, mais ensuite on a pu montrer relativement facilement, grâce à des réductions simples au problème SAT, la NP-complétude d'autres problèmes : on montre qu'il est aussi difficile de résoudre ces problèmes que de résoudre SAT.

Une autre façon de formuler la question $P = NP$? est donc la suivante : si on est capable de vérifier rapidement une solution à un problème de satisfaisabilité, c'est-à-dire en temps polynomial, est-on capable de déterminer rapidement également une solution ?